Assignment - 2

Page No.
Date

2] $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to (E) \mid id$

→ Recursive descent is a top down parsing technique that constructs parse tree from top & the input is read from L to R & recursively parses the input to make a parse tree.

→ A form of RD parsing that doesn't require any backtracking is known as predictive parsing. It has the capability to predict which production is to be used to replace input string

→ For the given grammar we first eliminate left recursion for E as

$E \to TE'$ $\quad E' \to +TE' \mid \varepsilon$

→ Then we eliminate L.R for T as

$T \to FT'$ $\quad T' \to *FT' \mid \varepsilon$

by adding new variables E' and T'

→ Thus the obtained grammar after eliminating LR is

$E \to TE'$, $E' \to +TE' \mid \varepsilon$, $T \to FT'$, $T' \to *FT' \mid \varepsilon$, $F \to (E) \mid id$

→ This grammar contains no left factoring

* Alg to eliminate L.R → Rules for no recursion $A \to A\alpha$, $A \to \alpha A'$, $A' \to \alpha A' \mid \varepsilon$

1] Arrange the non-terminals in some order A1, A2, ... An

2] for i := 1 to n do begin

for j = 1 to i-1 do begin

replace each production of form $A_i \to A_j\gamma$

by productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$.

where $A_j \to \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j-productions

end

eliminate the immediate LR among A_i productions.

end.

* Alg to eliminate L.F. → Rule for no factoring ② $A \to \alpha_1 A \mid \alpha_2 A \mid \alpha_3 A$

1] In L.F it isn't clear which of two alternative productions is used to expand a non-terminal (eg. A in given below grammar)

i.e. if $A \to \alpha\beta_1 \mid \alpha\beta_2$

2] we don't know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.

3] To remove this ambiguity we replace all A productions

variable (non T), ... that begins with same variable, in 1 (max steps.

containing a as prefix by $A \to \alpha A'$ then $A' \to \beta_1 | \beta_2$

* the construction of a predictive parser is aided by two functions associated with Grammar G. These functions are First and Follow

* Rules of First()

1) If X is a terminal, then First(X) is {X}
2) If $X \to \varepsilon$ is a production, then add ε to first(X)
3) If X is a non-terminal and $X \to a\alpha$ is a production then add a to first(X)
4) If X is non terminal and $X \to Y_1 Y_2 .. Y_k$ is a production, then place a in first(X) if for some i, a is in first(Y_i) and ε is in all of first(Y_1), first(Y_2), ..., first(Y_{i-1}), that is $Y_1, Y_2 .. Y_{i-1} \Rightarrow \varepsilon$.
If ε is in first(Y_j) for all $j = 1, 2, ..., k$, then add ε to first(X).

Given:

$\therefore$ first(E) = first(T) = first(F) = {(, id} .... (4)
$\therefore$ first(E') = {+, ε} .... (2) and (3)
$\therefore$ first(T) = first(F) = {(, id} .... (4)
$\therefore$ first(T') = {*, ε} .... (2) and (3)
$\therefore$ first(F) = {(, id} .... (2) and (3)

* Rules for follow

1) If S is start symbol, then follow(S) contains $
2) If there's production $A \to \alpha B \beta$, then everything in first(β) except ε is placed in follow(B)
3) If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where first(β) contains ε, then everything in follow(A) is in F(B)

Given:

$\therefore$ follow(E) = {$,)} .... (1) and ")" is R.H.S after E in rules
$\therefore$ follow(E') = {$,)} .... (3) $\therefore$ follow(E') = follow E as $E \to TE'$
$\therefore$ follow(T) = {$, +,)} .... follow(T) = first E' → (2) & as $E' \to \varepsilon$ $\therefore$ follow(T) = follow(E) (3)
$\therefore$ follow(T') = follow(T) = {$, +,)} .... (3)
$\therefore$ follow(F) = first T' = {*, $, +,)} → (2)
as $T' \to \varepsilon$ $\therefore$ follow(F) = follow T) → (3)

m[A, a] 1) for table at row of E and col id ... substituting the rule, E → TE' & the T → FT' & F → id
2) similarly we obtain parsing table for remaining rows & col.

5) * Parsing table for given grammar

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id				F→(E)	

* lets take example string id+id*id & check if it is accepted

Stack	Input	Output	Stack	Input	Output
$E	id+id*id$	E→TE'	$E'T'	*id$	T'→*FT'
$E'T	id+id*id$	T→FT'	$E'T'F*	*id$	pop
$E'T'F	id+id*id$	F→id	$E'T'F	id$	F→id
$E'T'id	id+id*id$	pop	$E'T'id	id$	pop
$E'T'	+id*id$	T'→ε	$E'T'	$	T'→ε
$E'	+id*id$	E'→+TE'	$E'	$	E'→ε
$E'T+	+id*id$	pop	$	$	Accepted
$E'T	id*id$	T→FT'			
$E'T'F	id*id$	F→id			
$E'T'id	id*id$	pop			